可用内存5: 12MB。

OI，第一阶段，7.10-4.11.2013

软管

一条蛇躺在一个3×n的板子上。蛇的连续片段被编号为1至3n。有连续数字的片段（即和2，2和3，3和4 ......）位于棋盘相邻的字段上。例如，在大小为3×9的棋盘上，一条蛇可能是这样躺着的。

7	6	5	4	17	18	19	20	21
8	1	2	3	16	15	26	25	22
9	10	11	12	13	14	27	24	23

蛇所占据的一些领域已被模糊化。你能重建蛇的布局吗？

输入

*标准输入的第一行包含一个整数n（1¬n¬1000），表示棋盘的长度。接下来的三行包含对棋盘的描述；第i行包含n个整数a_{ij}（0¬ a_{ij}¬ 3 n为1¬ j¬ n）。如果a_{ij}>0，那么a_{ij}表示位于棋盘第i行第j个区域的蛇形碎片的数量。另一方面，如果a_{ij}=0，那么位于所考虑的蛇形碎片的数量为
领域是未知的。*

在总分值为15%的测试中，条件n¬ 10发生，在总分值为40%测试中，条件n¬ 40发生，在总分值为70%的测试中，条件n¬ 300发生。

输出

你的 程序应该输出三行到标准输出。第i行应该包含n个正整数b_{ij}（对于1¬j¬n）。所有的b_{ij}数字加起来应该是1到3n的数字的排列组合。输出中的数字排列应根据输入数据再现蛇的可能位置。

你可以假设至少有一种方法可以在棋盘上再现蛇的位置。如果有一个以上的解决方案，你的程序可以输出其中任何一个。

例子

对于输入：其中

```
9
0 0 5 0 17 0 0 0 21
8 0 0 3 16 0 0 25 0
0 0 0 0 0 0 0 0 23
```

一个正确的结果是。

```
7 6 5 4 17 18 19 20 21
8 1 2 3 16 15 26 25 22
9 10 11 12 13 14 27 24 23
```

解决方案

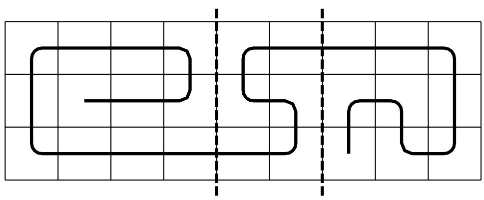
让我们首先尝试用图论的语言来表述任务的内容 的图形。董事会我们可以把$3\times n$设想为一个无向图，其顶点是棋盘的字段。顶点之间的边发生在对应于这些顶点是侧向相邻的。蛇在棋盘上的排列与该图中的某个*哈密尔顿路径相对应*，也就是说，该路径正好经过该图的每根柳条一次。

棋盘上的一些字段被成对分配给不同的数字，从1*到*3。字段号表示我们要找的是汉密尔顿路径中的哪个顶点。因此，我们要寻找任何*与顶点编号一致的*哈密尔顿路径。在任务的主体中，我们发现断言存在这样一条路径。

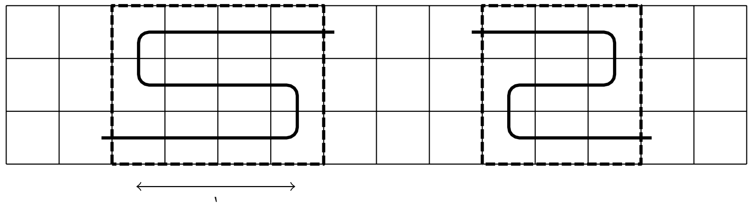
软管怎么能说谎呢？

为了更接近一个解决方案，看看不同的哈密尔顿路径在$3\times n$晶格上是什么样子是很有用的，现在不考虑顶点的编号。

让我们首先考虑这样一种情况：棋盘可以被沿网格线的垂直切口分割成碎片，在这些碎片之间路径只经过一次。下图显示了一个取自任务内容的这样一个路径的例子，有两个可能的切割位置。

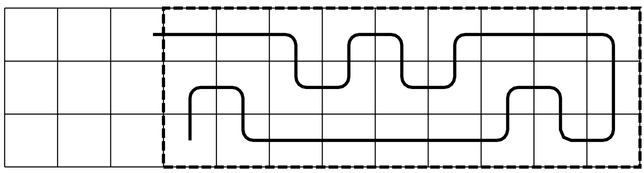


在所有这样的切割之后，整个棋盘的哈密顿路径坍缩为较短的棋盘碎片的哈密顿路径（在上面的例子中，这些是三条路径）。因此，我们有两个最外层的棋盘碎片和中间的一些棋盘碎片。后者有一个额外的条件，即其中路径的起点在片段的左侧，终点在右侧。画了一会儿后，可以看出这样的哈密尔顿路径一定是*"之"字形的*。

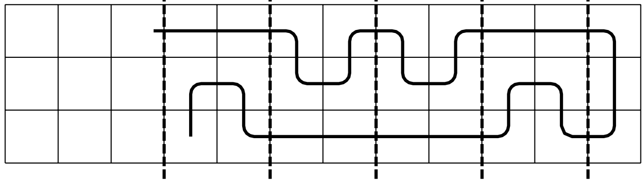


该图显示了 "之 "字形的唯一两种可能安排。之字形的第二个参数是它的长度（即水平方向上占据的字段数），表示为
在图*中以*`。在`=1的极端情况下，"之 "字形限制在其
板的一部分只是一个垂直部分。

我们仍然要分析棋盘最外面的两块碎片。在他们每个人的情况下，我们所知道的哈密尔顿路径是它的起点在片段的左边（分别是右边）。在任务正文中的例子中，恰好在棋盘的最右边的片段中，路径的另一端与该片段中的起点在同一侧。 我们会说这样一条路是一条*来回的*路。它有一个相当规则的形式，比如说

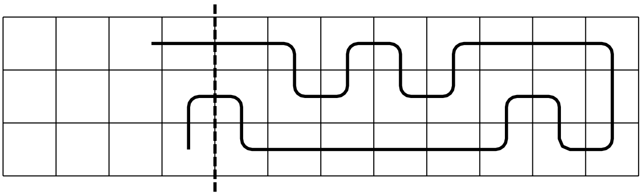


如何描述这样一条道路？我们可以把棋盘上的一段棋分成几对连续的柱子（不包括最后一列，路径在那里转回）。在每一个这样的配对中，顶部的四个字段或底部的四个字段都形成了 *一个驼峰*。



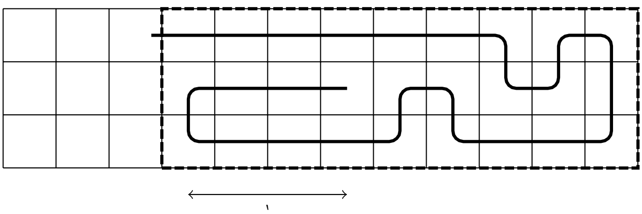
我们只需指出，如果摘录中的列数是偶数，最左边的驼峰就会被削减一半。

对我们来说，一个更方便的方法是看这样一个路径，如下所示。如果我们截断路径两端所在的那一列，结果就是一个较短的棋盘部分的汉密尔顿路径，也具有路径两端位于该部分的同一侧的特性（即也是一个*来回*的路径）。



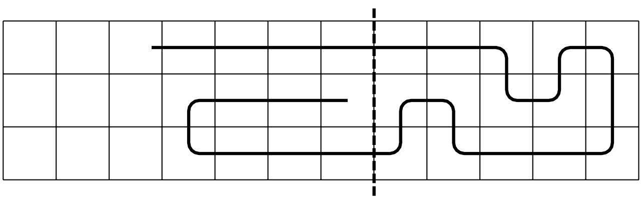
我们写这一切的前提是，汉密尔顿在极端片段的路径是一个*来回的*路径。如果路径在极端片段内的某个地方结束，那么为了到达该点，它必须首先返回到它开始的那一边（访问那里的棋盘片段的字段）。

事实上，如果路径不再需要返回初始面，这就意味着它必须以 *"之 "字形的方*式访问这些领域，然而，这将与分割后的棋盘部分是极端的事实相矛盾。到达最初的一侧后，道路向后转到尽头。由于我们只有三行可用，所以这里描述的回头相当于一个*绕行*，在这个*绕行的*过程中，路径向一边跑，另一边跑。

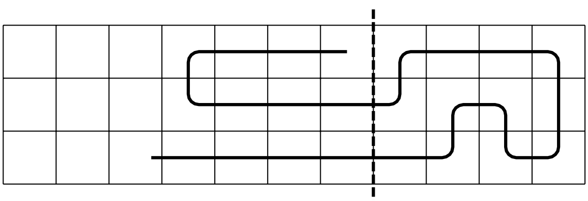


在图中，我们已经标出了*包覆*的长度`。那么*较链*后面是什么呢？在这里，我们只有一条哈密尔顿路径，其两端都在较短部分的同一侧--它正是一条*来回的*路径。

我们之前考虑过的!

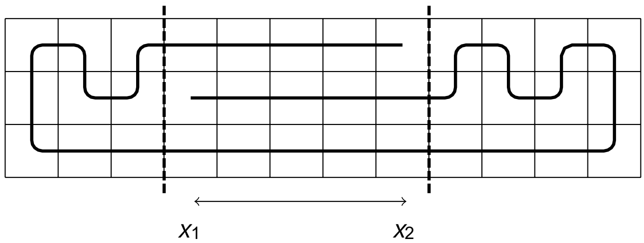


还有两件事值得注意。首先，*漩涡的*形状不一定要和上面的完全一样。只有 "回头 " 部分必须由两个连续的线条组成。这里是另一个完全正确的*包装*。



第二，可能会出现这样的情况，即漩涡贯穿整个极端部分的长度。在这种情况下，整个路径将具有本节开始时考虑的熟悉的*之字形*形式。

在最后，我们给自己留下了最后一种情况，你可能根本就没有注意到。这是一种汉密尔顿路径，不能在任何一点上被垂直切割成较短片段的汉密尔顿路径。如果
这样的路径从棋盘的左边缘或右边缘开始，它只是一个单一的*之字形*或单一的*来回*路径，有或没有*包络*。如果不是，事实证明，这样的路径只有一种类型，由*第二种类型的包裹*和两个*来回*类型的路径组成。



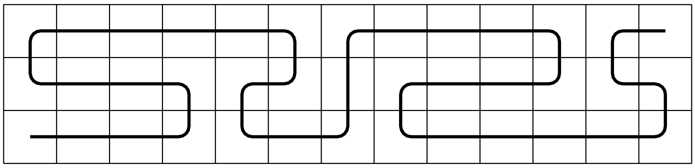
*第二种类型的包装*器有两个参数。 x_1 和 x_2，指定其开始和结束的位置。它包括两条只与棋盘的一部分相连的路径和一条连接棋盘两部分的路径；棋盘三行中的三条路径的所有安排都是可能的。我们将称这样的哈密尔顿路径为*扭曲的*路径。我们已经省略了精确的理由，即每一个
"不可分割 "的道路是一条曲折的道路。

综上所述，我们可以写出，3×*n* *格子*上的每条哈密尔顿路径要么是*扭曲的*，要么是由最多两条*来回*路径组成的，每条路径都可能有一个额外的*包裹物*，而任意数量的中间的 *"之 "字形*。

第一次安装

现在是时候提醒自己，我们的任务在某种程度上规定了我们要找到的哈密尔顿路径。也就是说，我们有一个给定的二维
一个数组*a*[1...*n*, 1...3]，指定路径中每个字段的数字，其中0
意味着该字段的编号不为人知（我们有意将该字段的尺寸调换为
阵列的内容与任务的内容有关，以便它们与坐标系的轴线对齐）。我们希望利用之前的组合学考虑，创建一个高效的算法，以寻找与给定场数相匹配的哈密尔顿路径。该算法的基础将是动态编程方法。

在本节中，我们将沉溺于一些简化，这将有助于我们抓住模型解决方案中包含的主要思想。 也就是说，我们将假设我们正在寻找的哈密尔顿路径只由*之字形*组成（这种路径的一个例子可以在下一页的图中找到）。在下一节中，我们将展示如何扩展这个解决方案以处理其他类型的路径。我们将以*复杂度为*$O(n^2)$的解决方案为目标。



我们将尝试应用自然的想法，通过从左到右增加更多的之字形来建立我们的汉密尔顿路径。如果我们已经覆盖了棋盘的一个部分，我们将尝试在下一个位置应用不同长度的之字形，看看它们是否与给定的字段编号相符。如果出现任何

这些之字形将与盒子的编号相对应，我们将记住我们已经成功地覆盖了棋盘的一个新的、更长的部分的信息。最后，我们将检查我们是否已经用其中一种方式覆盖了整个棋盘。

在算法的开始，我们仍然要对路径的起点做出一些决定。最左边的之字形可以从棋盘的左下角或左上角开始；路径上每个后续之字形的起点已经明确确定。此外，我们需要确定我们要从哪一边开始。

以及从汉密尔顿路径的哪一端开始，换句话说，哪个字段将被编号为1。

和哪个*3n*。在这个描述中，我们将假设第一个人字形开始于棋盘的左下角或左上角区域，我们想把数字1分配给它。

我们将写一个辅助函数

$$zigzag(x_1, x_2, y)_1$$

检查 $1 \neg x_1 \neg x_2 \neg n$ 和 $y_1 \in \{1, 3\}$*的*情况。棋盘上从第x_1 到第x_2 列的部分是否能正确地被一个人字形覆盖，这个人字形的起点是场$P = (x_1, y_1)$，终点是场$K = (x_2, 4 - y_1)$。这种形状的

人字形的设置是毫不含糊的。事实证明，--在刚才的假设下--人字形中的盒子的编号也是毫不含糊地设定的!事实上。

之字形的连续场的数字从场*P*上的数字$3x(_1 - 1) + 1$开始增加。

并以场*K*上的数字x3$_2$ 结束。

假设我们已经有了这个函数。 然后，我们可以使用一个逻辑值*覆盖的*

单一数组[0*...n,* 1*...*3]来实现整个解决方案，其中包括

covered[*i, j*]存储了我们是否已经成功地用黄金覆盖了棋盘的某一部分的信息。

*第一i*列的妻子通过以*fieldui*为终点的路径，（*j*）。我们假设整个数组在一开始就充满了假值。以下是伪代码。

```
covered[0 , 1] := covered[0 , 3] := true。
for i := 0 to n -1 do
    对于j∈{1, 3}做
        如果覆盖[i, j]，则开始
            for k := i + 1 to n do
                如果zigzag( i + 1 , k, )j那么
                    covered[ k, 4 - j] := true。
        结束
返回covered[ n, 1] 或 covered[ n, 3]。
```

如果我们能够在恒定时间内计算出*人字形*函数的值，上述伪代码将在期望的*时间*$O(n^2)$内工作。

然后让我们来处理 *"之 "字形*函数。如果我们注意到每个 "之 "字形由位于 "之 "字形的个别行中的三个*段*组成，我们就会省去很多工作。

董事会。具体来说，对于调用*zigzag*(x_1 , x_2 , y_1)，有以下几个部分。

- (x_1 , y_1)- (x_2 , y_1)的连续场数从$3x_1 - 2$到$2x_1 + x_2 - 2$。
- (x_2 , 2) - (x_1 , 2)，连续的字段号从$2x_1 + x_2 - 1$到$x_1 + 2x_2 - 1$。
- (x_1 , $4 - y_1$) - (x_2 , $4 - y_1$)，连续的字段号从$x_1 + 2x_2$ 到 $3x_2$ 。

对于其中的每一个部分，我们要不断地检查板框的预设编号是否与我们要分配的数字相符。我们将为

在一个额外的*段*数组的帮助下，我们将再次使用动态编程提前确定其值。为了适应$O(n^2)$时间和内存，我们需要相当巧妙地设计这个阵列。

我们将这样做。 因此，*段*[*x*，*y*，*k*，]*v*表示棋盘的连续方格数。从场（*x*，*y*）开始，沿着$k \in \{$*左*，*右*$\}$的方向走。，其数字为与编号v, $v+1$, $v+2$, ...一致。...，据此，该领域(*x*, (*y*)被重新编号为*v*。形式上，对于*k*=`right`，我们希望有。

$$段[x, y, k,]v = \max\{l \quad 0 : fits(a[x, y], (v) \wedge ... \wedge match(a[x + l - 1, y], v + l - 1)\}$$

据此，*fit*(*p*, *q*) 在$p = 0$或$p = q$时为真。对于
k = `left`，我们以类似的方式定义它。

以这种方式定义的数组有$O(n^2)$个字段，我们可以在$O(n^2)$个时间内将数值填入其中。只是重要的是不要迷失在其大量的尺寸中。如果*k*=`right`，我们从右到左确定数组的元素。

```
对于x := n downto 1 do
    for y := 1 to 3 do
        对于v :=1到3 n做
            如果不适合（a[x, y], v），那么
                节[ x, y,right,v]。:= 0
            否则
                节[ x, y,right,v]。:= 1 + 段[ x + 1 , y,right, v+ 1]。
```

这里我们假设对于$x = n + 1$，我们总是有一个*段*[*x*, *y*, *k*,]v= 0。
k=`left`的计算是类似的，只是从左到右。填好*段*数组后，我们计算*之字形*函数的结果，如下所示。

```
函数zigzag(x1 , x2 , y )1
开始
    返回（段[x1 , y1 , right, 3x1 - 2] x2 - x1 + 1）和
        (节[x2 , 2，左，2x1 + x2 - 1] x2 - x1 + 1）和
        (节[ x1 , 4 - y1 , right, x1 + 2 x2 ] x2 - x1 + 1）。
结束
```

模型解决方案

为了得到一个完整的解决方案，我们仍然需要，bagatelle，考虑$3\times n$格子上的哈密尔顿路径的所有其他配置。 幸运的是，最困难的部分实际上已经在我们身后了。

在我们详细研究了之字形的情况后，读者会很容易相信，我们能够以完全相同的方式在恒定的时间内检查每一个*结点*，以及汉密尔顿的扭曲路径中出现的*第二类结点*。

第二个重要因素是*来回的*路径。这些东西要么单独出现，要么有一个包装器，或者最后作为扭曲路径的一个组成部分。我们将为它们引入一个相当通用的逻辑值阵列 *tizp*[x, y_1 , y_2 , k,]v，其中的元素表示：*那里*是否有一个类型的路径，*以及从返回--的*路径。终端为字段(x, y_1)和(x, y_2)，包含棋盘的所有字段，位于在从最后的字段的方向k，分配字段(x，升序排列。我们有 y_1)数v和进一步的数字：$x\in\{1, \ldots, , ny\}_1$, $y_2 \in\{1, 2, 3\}$, y_1 =6 $v\in\{1, ..3n,\}$. y_2 , $k\in$\{`left`, *right*\}和

这个数组的大小为 $O(^{n2})$。为了及时填补它 $O(^{n2})$，我们将使用从前面的观察中可以看出，如果我们从一个被*来回*类型的路径覆盖的片段中移除一个极端的柱子（对于 k=右，它将是最左边的一列），我们得到一个短一列的片段的相同类型的路径。因此，为了确定k=`right`的数组的值，我们从

从右到左（$x = n, \ldots, 1$），每个类型的值*tizp*[x, y_1 , y_2 , k,]v，我们计算出基于一个或两个类型为*tizp*[x + 1 , y_0 , y_0 , k, ^{0} v]的值。选择了两个 1 2

值，我们只有在\{y_1 , y_2 \} = \{ 1, 3\}*的*情况下才会有。，这相当于决定*驼峰*应该往哪边走。对于k=`left`，我们进行类似的操作，只是计算从左到右进行。

这样一来，我们已经涵盖了路径的所有组成部分。最后，剩下的就是我们要把这一切放在一起。例如，我们将展示如何考虑由 *"之 "字形*和 "之 "*字形*组成的所有路径。 类型为*tam的路径和回来*。

```
for x := 1 to n do
    foreachy1 , y2 ∈{1, 2, 3}。, y1 =6 y2 do
        如果 tizp[ x, y1 , y2 , left, 1] 那么
            covered[ x, y2 ] := true。
covered[0 , 1] := covered[0 , 3] := true;
{ 用覆盖物填满数组的其余部分（人字形的动态编程） }
结果 := 假的。
for x := 1 to n do
    foreachy1 , y2 ∈{1, 2, 3}。, y1 =6 y2 do
        如果 tizp[ x, y1 , y2 , right, 3x -2] 和 covered[ x -1, y1 ] 则
            结果 :=真。
for y := 1 to 3 do
    结果 :=结果或覆盖[的结果 n, y]。
返回结果。
```

如果我们在上述伪代码中加入对*Tizp*路径中的*卷曲的*考虑（这可以用一个与*之字形*相似的函数来完成，使用一个*段*数组），我们得到的解决方案还没有考虑到只有*扭曲的*路径。后者在传统上会受到一些忽视的对待。

我们将让读者考虑如何处理这些问题。

在解决方案的最后，我们还剩下最后一个困难，与其说是概念性的，不如说是实施性的，即结果的重建。到目前为止，我们只寻求了一条路径是否存在的答案（这在其他方面是可以保证的），现在我们必须考虑如何找到这条路。在这里，我们将使用标准的方法，在动态编程中再现结果。我们将为解决方案中存储逻辑值的每个数组绑定一个额外的辅助数组，即所谓的父数组。如果在逻辑阵列的任何字段中 出现了一个

变异的

为真，辅助数组的相应字段将包含信息

关于我们确定这一数值的依据。例如，在一个*覆盖的数组的*情况下，这将是一对数字

，指定数组字段的索引 的基础上，我们在给定的字段中获得了真值（或由于类型的*来回*路径而出现的信息），而在数组*tizp*

的情况下，它可以

是一个类似的五个数字，指定数组中前一个字段的索引，或者只是一个数字0或1，表示最后一个*驼峰*的方向。在这两种情况下，我们都会记住相关信息，当 当我们现在想重建所产生的路径时，我们只需回溯父亲的值。

基准解决方案的实现可以在文件`waz.cpp`（记忆的恢复）和`waz1.cpp`（动态编程）中找到。文件`wazs2.cpp`和`wazs7.cpp`包含一个稍差的解决方案，复杂度为$O(n^3)$，但实现起来稍显简单。它省略了*段*数组，所有的*之字形*和*包裹*都是按字段顺序检查的。

其他解决方案

由于该任务与在图中寻找汉密尔顿路径的问题相似，在解决该问题时可以尝试使用标准技术。

最简单的指数式解决方案（`wazs1.cpp`）递归地构建了一条路径，从每个可能的顶点开始。这个解决方案在某些情况下效果很好，例如，当棋盘被完全填满时（在这种情况下，它是一个简单的搜索）或当它是空的时候。这种类型的解决方案在比赛中得到了20-30分。

另一种方法是利用任务中给出的图形具有较小的宽度（等于3）这一事实。这是动态编程，其中单个*状态*描述了图的横截面上的配置，即位于棋盘一列的三个顶点。在这样的状态下，我们假设我们已经把这个状态下的所有东西都相应地编号到了左边，而且我们记得所有与构建其余路径相关的信息。例如，这可以是以下信息。

- 棋盘的列号(x)。

- 我们在棋盘的这一列中输入了哪三个数字（三元素阵列*t*）。

- 到目前为止，黑板上使用的*A*数集是什么--不难看出，如果已经输入的数字可以扩展到解决方案，那么*A*数集就表达了
  是最多两个不相交的区间之和。
- 在考虑到这一列和之前的一切之后，所考虑的三种信念中的每一种都缺少多少个哈密尔顿路径上的邻居--每个顶点在哈密尔顿路径上最多有两个邻居（表*deg*）。

下面是电路板的一个部分，有相应的状态。

17	18	19	20	21
16	13	12	1	2
15	14	11	10	9

- $x = 5$
- $t[1] = 9$, $t[2] = 2$, $t[3] = 21$
- $A = [1, 2] \cup [9, 21]$
- $deg[1] = deg[2] = deg[3] = 1$ 预备 若

干 例子 的实现。 这个的 类型 的方法（`wazs[3-6].cpp`），它们在信息量上有所不同。 记忆中的单一
状态（以及动态编程的维度数量）和方式。
状态之间的转换。在所有的解决方案中，我们只记住实际允许的状态。根据优化情况，这些类型的解决方案在比赛中的得分在30分甚至60分之间。

仍然值得强调的是`wazb1.cpp`的解决方案，它超出了这项任务的内存限制。该基准解决方案使用不到200MB的内存。然而，这是由于本方案中使用的数组被粗略地声明，使用1字节和2字节的整数类型（即C++中的`char`和`short`）。
如果到处都使用4字节的类型（`int`），可能会消耗超过600MB的内存，这正是解决方案`wazb1.cpp`中发生的情况。任务中的内存限制（512MB）是以这样的方式选择的，即只需要基本的谨慎来适应。

测试

这项任务提供了相当多的测试，但没有分组，每一项都是 价值在1到4分之间
。 从某种意义上说，每个测试都可以被当作一个独立的难题。

在大多数情况下，测试是根据模型解决方案中的配置生成的，通过对电路板上的某些随机字段进行归零。最困难的测试的特点是有少量的字段与已知的蛇形碎片的数量（例如，一个*扭曲的*路径已经可以用三个字段的值来强制实现）。在其他情况下，恶意安排已知值的字段是为了帮助测试解决方案的有效性。例如，在一个*来回路径*的情况下，指定 *在后面，*只在顶行或只在底行指定字段值决定了整个配置，但如果从另一边开始，指数解会出错很长时间。最后，这组测试包括空的或几乎是空的棋盘和测试

棋盘完全被填满。